Testing

Group 18

Team B

Olivia Betts
Zac Bhumgara
Nursyarmila Ahmad Shukri
Cameron Duncan-Johal
Muaz Waqas
Oliver Northwood
Teddy Seddon

Over the course of our project, we used a combination of automated unit tests and manual testing. Generally, automated testing was more made use of for the classes for food and actors, whereas manual testing was used for classes for the different screens in the game and what they would render.

The first step we took was assigning roles to everyone in the group. In terms of testing, Muaz and Teddy were assigned to make and run the tests, as well as complete the testing report. We decided to use Junit 4 for our automated testing. Junit makes it easy for the machine to recognise the presence of a test due to the @Test keyword, as well as allowing us to use the *assert* keyword to check for preconditions.

The first stage was to create an environment in which we could start testing the original game. Therefore a GdxTestRunner class was made, which allowed us to run the tests using the Headless configuration. This was useful, because it meant that there was no actual graphics of the game being produced, and it would be easier to just test the logic.

Upon adding the correct dependencies, the first tests which we made were asset tests. These would check for the presence of assets essential to the running of the game in the assets folder. Therefore, they were very important for the overall usability of the project. We made asset tests for all of the assets related to the recipes and the customers, as well as for the music.

After the asset tests were complete, we moved onto logic tests. We started with testing the individual classes for correctness and usability e.g. Burger class, Customer class. This allowed for easy detection and correction of issues at class level. We would test all the methods in the class and make sure that they were all working as expected. After that, we planned to move onto testing some of the user input elements. Unfortunately, at this point, we came across a problem and that was that most of the code was in the GameScreen class, which had not made any use of lazy evaluation. This meant that the code was not very testable. Every time the GameScreen class was called, the spriteBatch was called alongside it (but spriteBatch could not be null) - for this reason, we struggled to run the Junit automated tests in Headless configuration. Our solution for this was to test most of the GameScreen class using manual testing. We did end up making some tests for the GameScreen class, but they did not run as expected. For this we also implemented Mockito. We were unable to automatedly test most of the UI elements.

We then moved onto manual testing. Using the requirements, we created tests for all of the requirements, as it is good practice to have tests for each requirement. For all the tests we created success and failure scenarios and tested them all, documenting thoroughly in a Word document, and adding pictures to explain the current state of the game.

Alongside all of this testing, we were implementing more code as well, to finish the game. Everytime one of the implementation team would commit something to the github, we would aim to make tests for it, to check that it is not altered in later commits. Another important thing we remembered was to comment and create JavaDocs for all of the code to make it easy to understand.

At the end of the testing, to make sure that we had covered every requirement, we created a traceability matrix, as well as a testing report. Both of these documents were useful in helping us to figure out the overall performance of our project for our tests.

In terms of the overall timing of the tests, we carried out most of the asset tests at the start of the testing cycle. This was to ensure that all necessary assets were present before proceeding with logic tests. The logic tests were then conducted during the development process, to double check if the new implementations are correct. The manual testing was done at the end of the development process, once the game was approaching completion.

Test reports

Testing material URLs

Software Testing Report [22 marks]:
a) Briefly summarise your testing method(s) and approach(es), explaining why these are appropriate for the project. (5 marks, 1 page)
b) Give a brief report on the actual tests, including statistics of what tests were run and what results were achieved, with a clear statement of any tests that are failed by the current implementation. If some tests failed, explain why these do not or cannot be passed and comment on what is needed to enable all tests to be passed. If no tests failed, comment on the completeness and correctness of your tests instead (12 marks, 3 pages).
c) Provide the precise URLs for the testing material on the website: this material should comprise the testing results and coverage report generated by your automated testing tooling, and descriptions of manual test-cases that you designed to test the parts of the code that could not be covered by your automated tests (5 marks).

It is good testing practice to have tests created for all of the requirements for the project. Therefore as our first step for the testing, we looked through the requirements and brainstormed what tests could be relevant for each.